Optimizing Code 17

Optimizers edit your compiled code to make it more efficient. This chapter explains how optimizers work and how you use the THINK C optimizer.

Contents

What an Optimizer Is 343
- Why use an optimizer 343
- When not to use an optimizer 343

Using the Global Optimizer 344
- Induction variable elimination 345
- CSE elimination 345
- Code motion 346
- Register coloring 346

Using Local Optimizations 347
- Defer & combine stack adjusts 347
- Suppress redundant loads 348

17 Optimizing Code

What an Optimizer Is

An optimizer makes your code more efficient. The THINK C optimizer can speed up your program by as much as 20%, while it also makes the program smaller.

THINK C uses two types of optimizations: local and global. When you use a local optimization, THINK C looks at each statement being compiled and decides on the best way to implement it. For example, it could look at a variable declaration and decide to store it in a register, or it could look at a function call and decide on the quickest way to call it.

When you use a global optimization, THINK C adds a step after compiling a function to edit the compiled code. When you compile your program, the compiler produces something called intermediate representation. The optimizer edits the internal representation to make it quicker and smaller. It could eliminate steps or reorganize them. The optimizer then passes this revised internal representation to the code generator, which actually produces machine instructions.

Why use an optimizer

You can edit your own program, and you might even make it as efficient as an optimizer could. But letting the optimizer edit it instead has these advantages:

- Time savings. Tuning up old code takes time away from writing new code.
- Legible code. To make code more efficient, you might need to rewrite it in a way that is hard to read.
- Portable code. Code that's tuned for one type of computer (such as a Macintosh) may be inefficient on another (such as an IBM PC).

When *not* to use an optimizer

Use an optimizer when you're making the final build of a program you plan to use several times. Here are some programs you *don't* want to optimize:

- A program used only a couple of times. Using an optimizer can double your compilation time. It doesn't make sense to spend more time optimizing it than you'll spend using it.

For more information, see Chapter 12, "Debugging Programs."

- A program you're debugging. An optimizer generates machine code that differs significantly from your source code. Also, the global optimizer adds an additional pass over your compiled code and may double your compilation time.

Using the Global Optimizer

You can test and change the settings of these options from your code. See "Accessing Option Settings in Your Code" and "Global optimizer options" in Chapter 15.

To use the global optimizer, turn on the Use Global Optimizer option on the Code Optimization page of the **THINK C options** dialog. The options below Use Global Optimizer let you choose which optimizations you want THINK C to apply. The rest of this section describes each optimization in detail.

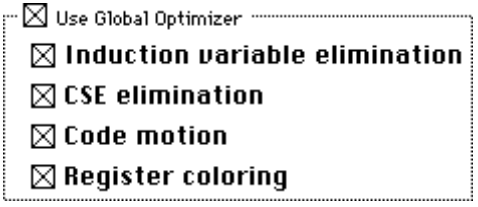

Figure 17-1 Use Global Optimizer option

Note

While optimizing an extremely large and complex function (one that contains several pages of dense code), the optimizer may not update the **Progress** dialog for several minutes. The THINK Project Manager has not crashed. The optimizer updates the **Progress** dialog when it's finished with the function.

The "Using Local Optimizations" later in this chapter describes other optimizations that don't require the global optimizer.

Induction variable elimination

This optimization makes loops faster, especially those that cycle through an array. It may make your code slightly larger. For example, take this loop:

```
int a[ARRAY_SIZE], i;

for (i=0; i<ARRAY_SIZE; i++)
   a[i] = GetNextElement();
```

Without this optimization, your code has to perform a multiplication each time it figures the address for the next array element (`i * sizeof(int)`). With this optimization, it remembers the address of the last element and just adds the size of an element to that address.

Use this optimization if your code contains many loops. It's especially useful on a Macintosh with a MC68000, like a Macintosh Classic or Plus, which performs 32-bit multiplication in software.

CSE elimination

This optimization makes your code smaller and faster. It replaces subexpressions that are used more than once with a temporary variable set to the subexpression's value. For example, look at this code:

```
a = i*2 + 3;
b = sqrt(i*2);
```

With this optimization, the optimizer assigns `i*2` to a temporary variable and computes it only once. It's as if the code were written like this:

```
temp = i*2;
a = temp + 3;
b = sqrt(temp);
```

Code motion

This optimization makes your loops faster. It moves expressions out of a loop that remain constant in each iteration. For example, look at this loop, in which `x` is a local variable:

```
while ( !feof(fp) )
   DoSomething( fp, x*5 );
```

With this optimization on, your code moves `x*5` outside the loop and computes it only once. It's as if the code were written like this:

```
temp = x*5;
while ( !feof(fp) )
   DoSomething( fp, temp );
```

Use this optimization if your code has a lot of loops.

Register coloring

This optimization makes your code smaller and faster. It searches your code for variables of the same type that are never used at the same time and has them share a register, if one is free. For example, take this function:

```
void foo(void)
{
   int i, j, k;

   for (i=0; i<10; i++);
      printf( "%d\n", i );

   for (j=2; j<1024; j+=2);
      printf( "%d\n", j );

   for (k=10; k>0; k--);
      printf( "%d\n", k );
}
```

Notice that `i`, `j`, and `k` are never used at the same time. With this optimization on, THINK C has them share a register.

Using Local Optimizations

You can test and change the settings of these options from your code. See "Accessing Option Settings in Your Code" and "Other optimization options" in Chapter 15.

This section describes local optimizations that the compiler performs. You turn them on in the left column of the Code Optimization page of the **THINK C options** dialog.

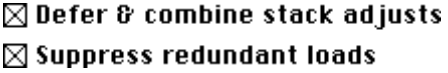

Figure 17-2 The other optimization options

To let THINK C optimize register allocation, see "Using register variables" in Chapter 15.

To learn about the global optimizer, see "Using the Global Optimizer" later in this chapter.

Defer & combine stack adjusts

If this option is on, THINK C saves time and space in code that calls many functions in a row.

If this option is off, THINK C follows this function-calling rule: When a statement calls a C function, that statement pushes the arguments on the stack before calling the function and pops them off after it returns.

If this option is on, THINK C changes that rule. If your code contains a run of C functions, it doesn't pop a function's arguments off the stack immediately after one of the functions return. It continues to call the other functions and to push their arguments on the stack. It lets the arguments accumulate and keeps track of how many are on the stack. When the run of functions is done, it pops the arguments off all at once.

THINK C can't use this optimization with Pascal functions such as Macintosh Toolbox functions or functions declared `pascal`. A Pascal function pops its arguments off the stack itself, before it returns.

Warning

If you're debugging and you use the **Skip to Here** command, you may want to turn this option off. When this option is on, THINK C must keep track of how much is currently on the stack. If you skip to a statement where that amount is different from where you are, THINK C will pop off the wrong number of arguments and may crash.

Suppress redundant loads

If this option is on, THINK C doesn't load data into a register when that data is already in a register. This optimization makes your code smaller and faster. The factory setting is on.

To understand how this optimization works, look at this example:

```
int j=0, i=1, k;

j = i + 1;
k = j;
```

When you reach the last statement, (`k = j`), the value of `j` is in two places: a register and in memory. If this option is off, `k` gets the value from memory, requiring you to compute the memory address. If this option is on, `k` gets the value from the register, saving you the time and space the computation takes.

THINK C loads `j` from memory into a register twice, once for each time it appears. If this option is on, THINK C loads it from memory only once.

If you're debugging, you may want to turn this option off. When you set a variable in a Data window, the debugger puts the new value into memory. However, when this option is on, your program may be using a value in a register and not in memory.

For example, look at the code above. After `j = i + 1` executes, there are two copies of `j`, one in memory and one in a register. Your program is using the register copy. The value in the register and in memory is `2`. If you examine `j`, the data window shows `2`.

Now, say you change the value of `j` in the data window to `10`. When your program continues, `k` is set to 2, anyway. The data window changed the value of `j` in memory, but your program used the value of `j` currently in the register.